

**Kennesaw State University**  
**College of Computing and Software Engineering**  
**Department of Computer Science**

3305 Section 01 Data Structures

Big O Analysis Version 1

2/23/2020

## **Basic Definition of Big O Notation:**

To understand the basis of what each Big O function does, there should be a basic understanding of what the textbook definition of Big O. According to Rob Bell, “Big O notation is used to describe the performance or the complexity of an algorithm.” (Bell “A Beginners Guide”). With that being said there are seven different types of Big O analysis:  $O(\log(N))$ ,  $O(N)$ ,  $O(1)$ ,  $O(N \log(N))$ ,  $O(N^2)$ ,  $O(2^N)$ ,  $O(N!)$ . According to bigocheatsheet.com, there is a graph that shows the performance of each Big O function varying from awful to excellent in terms of use/performance. Those that are awful are  $O(N!)$ ,  $O(2^N)$ , and  $O(N^2)$ . In the middle, still considered bad, is  $O(N \log(N))$ . And the rest Big O notations are fair, in which is  $O(N)$ , and good/excellent, in which are  $O(\log(N))$ ,  $O(1)$ .

## **Finding the Big O Analysis Using Math:**

When calculating Big O, it's good to know a few pointers on how to get the right Big O analysis. According to hackernoon.com, there are three factors that a programmer must consider when calculating Big O, in which are: Assume the worst, Remove Constants, and drop non - dominate terms (McHardy “Big O For”).

### **Assuming the worst:**

When you are assuming the worst, you want to the worst case of that function.

### **Remove Constants:**

If you got math that came out as:  $O(1 + 1 + 1) + O(N)$ , you really would get  $O(3N)$  but in Big O terms, you take off the constant and just get  $O(N)$ . By dropping the constant, the programmer will get it's most simplified version of Big O.

### **Drop Non – Dominate Terms:**

For Big O, there is weight system that takes place. If you got math that came out to  $O(N + N^2)$ , the final big O analysis will be  $O(N^2)$ , because  $O(N^2)$  holds a lot more weight than  $O(N)$ .

## **Average case vs. Worst Case vs. Best Case:**

When writing functions, the programmer must know the time complexity for the algorithm; and that can be achieved by looking at the three cases: Average case, best case, and worst case (Geeks for geeks “Analysis of Algorithm”).

### **Worst Case (Usually Done):**

For this case, you calculate the maximum number of operations to be executed within the algorithm written. In other words, you find the worst possible outcome and work your way from the bottom up. When looking at the upper bound, maximum number of operations, you have a guarantee of the running time.

### **Average Case (Sometimes Done):**

When you can find the average case, its best to use this because that is the average time complexity of the whole algorithm/function. Here is the math: sum all of the calculated values and divide the sum by the total number of inputs.

### **Best Case (Rare):**

The best case looks at the lower bound, in which is the least amount of operations needed to execute the algorithm, When looking at the lower bound, there is no guarantee of what the lower bound will be so it's almost impossible to find the best case.

## **What is Time complexity?**

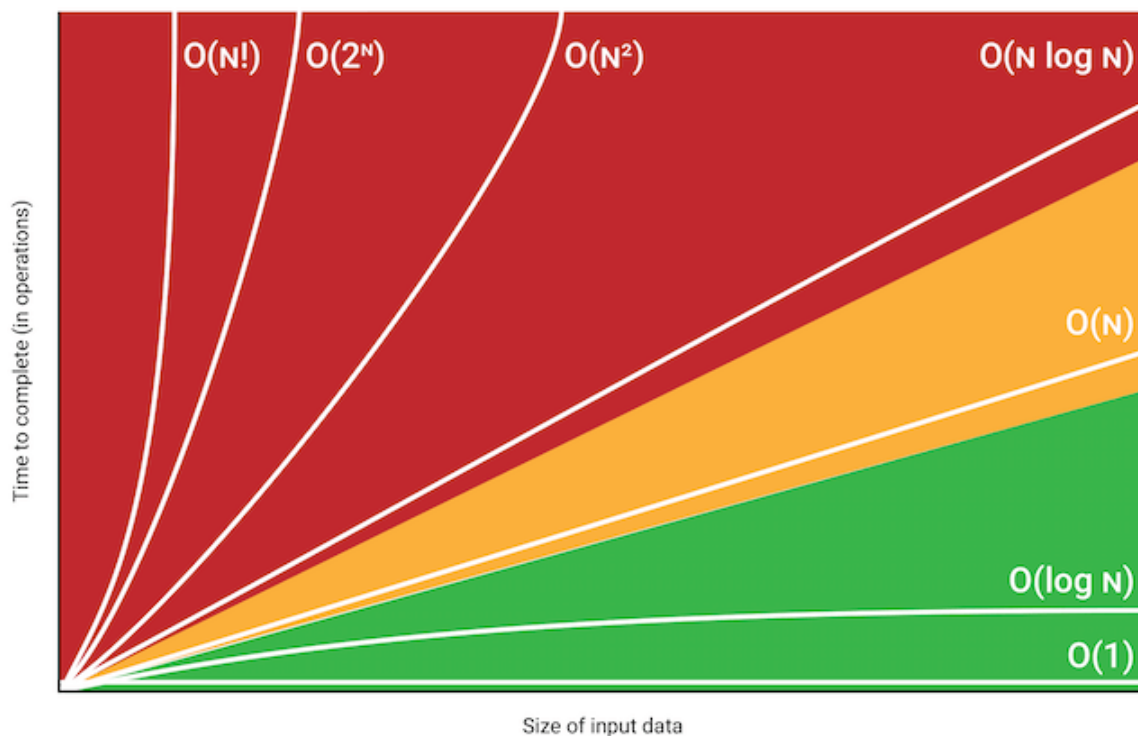
According to Stuart Kuridijan, time complexity is the amount of time an algorithm takes to complete a task (Kuridijan “Algorithm Time Complexity”). There are few different types of time complexity: constant complexity, linear complexity, quadratic complexity, exponential, and logarithmic complexity. Varying in completion time, each Big O function will have their own time complexity. In order to be efficient while writing code, the programmer must take into account the worst-case time complexity, the most time taken to complete the task, and work from the bottom up to reach the best-case, in which requires the least amount of time to complete the task. But sometimes the programmer can't reach the best case so they must resort to work with the average case, in which is the average amount of time to complete the task.

## What is Space Complexity?

Space complexity of an algorithm is the amount of space to run as a function of its input length,  $n$ ; consisting of space used by the input and Auxiliary space (“Time Complexity vs.”). Auxiliary is the extra space used by the algorithm while its being executed and space used by the input is the space that is being used during runtime within the length of the input. When programming, it’s best for the programmer to take into account the best time-space complexity when writing a program.

## Describing Each Big O Function:

To understand each individual Big O notation, I will explain using definitions. Therefore, I will start from the most efficient to the least efficient notation:



(Image #1) Graph Explaining the efficiency of each Big O notation.

- $O(1)$ , in which is the fastest notation, and also known to execute in the same time regardless the size of the input data set. No matter if there is 1 item vs 100 items, the program will always run in the same amount of time. That’s why  $O(1)$  is given the name, constant complexity.
- $O(\log(N))$ , also known as logarithmic complexity; in which means that the calculation time increases linearly as input data increases exponentially. For example, it takes

1 second to calculate 10 items, 2 seconds to calculate 100 items, and 3 seconds to calculate 1000 items.

- $O(N)$ , also known as linear complexity; in which means the calculation time will increase at the same pace as the input. For example it takes 1 second to calculate 1 item, 10 seconds to calculate 10 items, and 100 seconds to calculate 100 items.
- $O(N\log(N))$ , also known as divide and conquer because you divide the data into two halves making each sublevel to be divided in factors of 2 : 2,4,8,16, and so on... Because you do that, the time complexity is in the bad part of the graph above.
- $O(N^2)$ , also known as quadratic complexity; in which means the performance is proportional to the square size of the input data. You see this when there are nested For loops, in which can result in  $n^3$ , and  $n^4$  depending on how deep the nested iterations go. In consideration of time, 1 item will take 1 second, 10 items will take 100 seconds and 100 items will take 10,000 seconds.
- $O(2^N)$ , in which is exponential but doubles to each input added to the data set. Making the curve a lot more steeper than  $O(N^2)$  because its to the power of how much data and with the base of 2.
- $O(N!)$ , also known as factorial complexity making this the slowest Big O notation. For instance, if there are 5 elements in the input set, the equation for a factorial is will be  $5 \times 4 \times 3 \times 2 \times 1 = 120$  options. Therefore, because there are 120 options the run time will be very slow and almost hard to use because once you have 100 values there are  $9.332 \times 10^{157}$  options.

### **Big O Notation Using Data Structures:**

A data structure is a structure used to organize and store data in the most efficient way. The use of data structures is important because each data structure has its own functionality and flexibility. When writing programs using data structures, its important to see how efficient one does the job; so, when using Big O analysis, the programmer will know both the time complexity and space complexity of the algorithm. For a better grasp, I will show the Big O analysis on the basic data structures learned in this class in which are: Arrays, Linked List, Stacks, Queues, and Binary Trees. Each data structure will do the same operations, in which are: adding, removing, inserting, and searching. To see how efficient each data structure is with the following operations, I will provide syntax with a summary explaining the Big O analysis.

## Arrays + Insertion Sort + Linear Search + Binary Search + Merge Sort:

For an Array, I chose to do a dynamic array where there is no set capacity. The reason why I chose this is because you will need to insert the element in the desired index and move all the data back by one index.

```
Array::Array()
{
    index = 0;
    capacity = 0;
}
```

Right here I declared a default constructor, setting the index to 0 and the capacity to 0. For the code, index represents the current size of the array and the capacity represents the initial capacity of the array.

```
Array::Array(int size)
{
    capacity = size;
    index = 0;
    array = new int[capacity];
}
```

Array (int size) is the constructor that sets the initial size of the array.

```
Array:: ~Array()
{
    delete[] array;
}
```

~Array() represents the delete constructor. That is useful when there is allocated memory that needs to be deleted.

```
bool Array::search( int data)
{
    for (int i = 0; i < index; i++) //O(N)
    {
        if (data == array[i]) //O(1)
        {
            return true; // Item Found
        }
    }
    return false; // Item Not Found

    //Analysis: O(N)
}
```

Here is the **search function** for the Array.

**Postcondition:** The function will return true if the item is found or return false if the item isn't found.

**Analysis:**  $O(N)$  because, there is a linear search that is searching through the array linearly based on the size of the index. Then it checks if the data = array[i] to look at the value in array compared to the data taken in as the parameter.

```
void Array::add(int data)
{
    if (index + 1 > capacity) //O(1)
    {
        capacity++;
        int* tmpArray = new int[capacity];
        for (int i = 0; i < index; i++) //O(N)
        {
            tmpArray[i] = array[i];
        }

        delete array;
        array = tmpArray;
    }

    array[index] = data; //O(1)
    index++;
    //Analysis: O(1)
}
```

Here is the **add function** for the Array.

**Precondition:** Check to see if the index + 1 > capacity to protect the array from growing past the initial capacity.

**Postcondition:** If the precondition is true, there will be a temporary array with capacity + 1. Making the for loop activate, and linearly searching through the index of the array setting the temp array value[i] = array [i]. Once the temp array is populated, array will be deleted and set equal to temp array. Else, array[index] = data. Then, index will be incremented by 1.

**Analysis:**  $O(1)$

```
void Array::insert(int idx, int data)
{
    if (index + 1 > capacity) //O(1)
    {
        capacity++;
        int* tmpArray = new int[capacity];
        for (int i = 0; i < index; i++) //O(N)
        {
            tmpArray[i] = array[i];
        }

        delete array;
        array = tmpArray; //O(1)
    }

    for (int x = index; x >= idx; x--) { //O(N)
        array[x+1] = array[x];
    }
    array[idx] = data;
```

```

    index++;
    //Analysis: O(N)
}

```

Here is the **insertion function** for the Array.

**Precondition:** Check to see if the index +1 > capacity to protect the array from growing past the initial capacity.

**Postcondition:** If the precondition is true, there will be a temporary array with capacity + 1. Making the for loop activate, and linearly searching through the index of the array setting the temp array value[i] = array [i]. Once the temp array is populated, array will be deleted and set equal to temp array. After the temp array is populated, the array will then go through another for loop in which sets moves the values back from index to idx. Working backwards is the way I thought about this for loop; hence is why you see x--. When the values are moved in the correct order, array[idx] = data. Then increment index by 1.

**Analysis:** If the for loop is initiated, the big O analysis will be O (N). Here is the math:  $O(N) + O(N) + O(1) = O(N)$ .

```

void Array::delete_Element(int idx)
{
    assert(index != 0);
    for (int x = idx; x < index; x++) //O(N)

    {
        array[x] = array[x + 1]; //O(1)
    }
    index--;

    //Analysis: O(N)
}

```

Here is the **delete function** for the Array.

**Precondition:** Check to see if the index isn't 0. If so, the assert will catch invalid index and return with an assert method.

**Postcondition:** At the index where you want to delete, idx, and to the size of the array, index, Array[x] = array[x+1]. When each element is added in the right place and the index is deleted, index will decrease in size by 1.

**Analysis:** The big O analysis will be O (N) going through the for loop

```

void Array::insertionSort() // https://www.tutorialspoint.com/cplusplus-program-to-
implement-insertion-sort
{
    int key, j;
    for (int i = 1; i < index; i++) { //O(N)

        key = array[i]; //take value
        j = i;
        while (j > 0 && array[j - 1] > key) { //O(N)

            array[j] = array[j - 1];
            j--;
        }
        array[j] = key; //insert in right place //O(1)
    }
}

```



```

    }
}
//Analysis: O(N^2)

```

Here is the **Insertion Sort function** for the Array.

**Postcondition:** There is a key and j values created. The first loop will go through the array and set  $j = i$  and  $key = array[i]$ . Then the while loop will be initiated finding the smallest value and inserting it at the beginning of the array, and then decrease  $j$  by 1. Once executed  $array[j] = key$ .

**Analysis:** If the for loop is initiated, the big O analysis will be  $O(N^2)$ . Here is the math:  $O(N) * O(N) * O(1) = O(N^2)$ .

```

void Array::printArray()
{
    for (int i = 0; i < index; i++) //O(N)
    {
        cout << array[i] << " ";
    }
    //Analysis: O(N)
}

```

Here is the **Linear Search function** for the Array.

**Postcondition:** Simple Linear search function that prints out each element in the array.

**Analysis:** When the for loop is initiated, the big O analysis will be  $O(N)$ .

```

void Array::binarySearch(int target) //https://beginnersbook.com/2017/12/c-program-for-
binary-search/
{
    int first = 0;
    int last = index - 1;
    int middle = (first + last) / 2;
    while (first <= last) //O(N)
    {
        if (array[middle] < target) //O(1)
        {
            first = middle + 1;
        }
        else if (array[middle] == target) //O(1)
        {
            cout << target << " found in the array at the location " << middle +
1 << "\n";
            break;
        }
        else {
            last = middle - 1;
        }
        middle = (first + last) / 2;
    }
    if (first > last) //O(1)
    {
        cout << target << " not found in the array";
    }
}
//Analysis: O(Log N)

```

Here is the **Binary Search function** for the Array.

**Postcondition:** For a binary search, there will be three variables: first, last, and middle. While,  $\text{first} \leq \text{last}$ , in which is a  $O(N)$  function. When inside the while loop, the if statement checks if  $\text{array}[\text{middle}] < \text{target}$ , then has another else if and else statements checking the case of the target. Each have a  $O(N)$  time complexity.

**Analysis:** Big O analysis will be  $O(\log N)$

```
void Array::Merge(int low, int high, int mid)
{
    // We have low to mid and mid+1 to high already sorted.
    int i, j, k;
    int* temp = new int[high - low + 1];
    i = low;
    k = 0;
    j = mid + 1;

    // Merge the two parts into temp[].
    while (i <= mid && j <= high)
    {
        if (array[i] < array[j])
        {
            temp[k] = array[i];
            k++;
            i++;
        }
        else
        {
            temp[k] = array[j];
            k++;
            j++;
        }
    }

    // Insert all the remaining values from i to mid into temp[].
    while (i <= mid)
    {
        temp[k] = array[i];
        k++;
        i++;
    }

    // Insert all the remaining values from j to high into temp[].
    while (j <= high)
    {
        temp[k] = array[j];
        k++;
        j++;
    }

    // Assign sorted data stored in temp[] to a[].
    for (i = low; i <= high; i++)
    {
```

```

        array[i] = temp[i - low];
    }
    delete[] temp;
}

// A function to split array into two parts.
void Array::MergeSort(int low, int high)
{
    int mid;
    if (low < high)
    {
        mid = (low + high) / 2;
        // Split the data into two half.
        MergeSort(low, mid);
        MergeSort(mid + 1, high);

        // Merge them to get sorted output.
        Merge(low, high, mid);
    }
}
//Analysis: O(N)

```

Here is the **Merge Sort function** for the Array.

**Postcondition:** This merge sort function requires another function that merges the array when split into two arrays. In the merge sort function there will be a high and low taken in as the argument, and a middle created. Using those three variables will enable the merge function to be called. Inside the merge function is a while loop that works if the argument is meant. Then outside the while loop there is another while loop doing the same action but while  $i \leq \text{mid}$ . Finally, there is another while loop where  $j \leq \text{high}$ . Since there are three while loops the math will be  $O(N) + O(N) + O(N) = O(3N)$ . Drop the constant and the answer will be  $O(N)$ .

**Analysis:** If the for loop is initiated, the big O analysis will be  $O(N)$ .

## Linked List:

```

void list_head_insert(node*& head_ptr, const node::value_type& entry) {
    node* new_node = new node();
    new_node->set_data(entry);
    new_node->set_link(head_ptr);
    head_ptr = new_node;
}
//Analysis: O(1)

```

Here is the **add function** for the Linked List.

**Postcondition:** Linked List rely on pointers, so here I decided to add to the head of the linked list using the toolkit provided from our labs. So as the parameters, there will be a head\_ptr and an entry taken in. Using the big O analysis, the function will be running at constant time complexity. Therefore, you will be adding one value per function call.

**Analysis:** Big O analysis will be  $O(1)$

```

void list_insert(node* previous_ptr, const node::value_type& entry) {
    node* new_node = new node(entry);
    new_node->set_link(previous_ptr->link());
    previous_ptr->set_link(new_node);
//Analysis: O(1)

}

```

Here is the **insert function** for the Linked List.

**Postcondition:** So as the parameters, there will be a previous\_ptr and an entry taken in. Using the big O analysis, the function will be running at constant time complexity. Therefore, you will be adding one value per function call. Very similar to the add function; also, you won't need to worry about adjusting the size of this linked list because the last pointer will be pointing to NULL.

**Analysis:** Big O analysis will be O(1)

```

node* list_search(node* head_ptr, const node::value_type& target) {
    node* trav = head_ptr;
    // the first part of the condition guards the second part
    while ((trav != NULL) && (trav->data() != target)) {
        trav = trav->link();
    }
    // examine the results
    // if trav is NULL, we didn't find it, so return NULL
    // if trav is not NULL then trav->data() == target, so return trav
    return trav;
//Analysis: O(N)

}

```

Here is the **search function** for the Linked List.

**Postcondition:** For the linked list, I chose to linearly search through the list. By calling the while loop, the function will be running at linear complexity. Only function for the linked list that has a O(N) time complexity.

**Analysis:** Big O analysis will be O(N)

```

void list_head_remove(node*& head_ptr) {
    assert(head_ptr != NULL);
    node* tmp = head_ptr; /// first node, soon to be removed
    head_ptr = head_ptr->link(); // head_ptr now points at the second node
    delete tmp;
//Analysis: O(1)

}

```

```

void list_remove(node* previous_ptr)
{

```

```

node* remove_ptr;

remove_ptr = previous_ptr->link();
previous_ptr->set_link(remove_ptr->link());
delete remove_ptr;
//Analysis: O(1)
}

```

Here is the **delete function** for the Linked List.

**Postcondition:** For the linked list, there are two ways you can delete an item, and that's removing from the head or removing from the previous pointer. Both functions end up having the same time complexity and both allocate memory from the heap, in which soon get deleted.

**Analysis:** Big O analysis will be  $O(1)$

## Stack:

```

void stack::push(int x)
{
    if (isFull()) // O(1)
    {
        cout << "OverFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << x << endl;
    arr[++top] = x; // O(1)
// Analysis: O(1)
}

```

Here is the **add/insert function** for the Stack.

**Precondition:** Check to see if stack is full

**Postcondition:** For a stack, the push function is the way a programmer adds elements to the stack. First in, first out is the saying of how a stack works. When adding an element, the Big O analysis is  $O(1)$ .

**Analysis:** Big O analysis will be  $O(1)$

```

// Utility function to pop top element from the stack
int stack::pop()
{
    // check for stack underflow
    if (isEmpty())
    {
        cout << "UnderFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }
}

```

```

    cout << "Removing " << peek() << endl;

    // decrease stack size by 1 and (optionally) return the popped element
    return arr[top--];
// Analysis: O(1)
}

```

Here is the **delete function** for the Stack.

**Precondition:** Check to see if stack is empty

**Postcondition:** Pop is another way of deleting an element from a stack. You will delete the top element of the stack making the Big O analysis  $O(1)$ . Pop has the same time complexity as the push method because it depends on one element per function call; in which is saying the function is always at constant time.

**Analysis:** Big O analysis will be  $O(1)$

```

bool stack::search(int target)
{
    for (size_t i = 0; i < capacity; i++) //O(N)
    {
        if (target == arr[i]) //O(1)
        {
            return true;
        }
    }
    return false;
// Analysis: O(N)
}

```

Here is the **search function** for the Stack.

**Postcondition:** I implemented a linear search, also linear complexity, that searches from the bottom to the last element of the stack. Therefore, the big O analysis will be  $O(N)$ .

**Analysis:**  $O(N)$

## Queue:

A Queue is similar to a stack but the add and delete functions are different in syntax, but have the same Big O analysis. In a queue, there is saying “first in, last out”. In which means, you will in the front of the queue (enqueue) and delete (dequeue) at the end of the queue. Therefore, the Big O analysis for the stack and queue will be the same for the **add, delete, and search functions**.

The source code is below to see what each function looks like:

```

void queue::dequeue()
{

```

```

// check for queue underflow
if (isEmpty())
{
    cout << "UnderFlow\nProgram Terminated\n";
    exit(EXIT_FAILURE);
}

cout << "Removing " << arr[front] << '\n';

front = (front + 1) % capacity;
count--;
// Analysis: O(1)
}

// Utility function to add an item to the queue
void queue::enqueue(int item)
{
    // check for queue overflow
    if (isFull())
    {
        cout << "OverFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << item << '\n';

    rear = (rear + 1) % capacity;
    arr[rear] = item;
    count++;
// Analysis: O(1)
}

bool queue::search(int target)
{
    for (size_t i = 0; i < capacity; i++)
    {
        if (arr[i] == target)
            return true;
    }
    return false;
// Analysis: O(N)
}

```

## Binary Search Tree:

A binary search tree is a data structure that has an  $O(\log N)$  time analysis for all methods: **search, add, insert and delete**. The binary search tree has two sub trees; one left and one right. The left sub tree will be the values less than or equal to the root and the right values will be greater than the root's value.

```

void btree::destroy_tree(node* leaf)
{
    if (leaf != NULL) {
        destroy_tree(leaf->left);
        destroy_tree(leaf->right);
        delete leaf;
    }
    // Analysis: O(Log N)
}

```

Here is the **delete function** for the Binary Search Tree.

**Precondition:** Check if the leaf is not a NULL

**Postcondition:** Based on the leaf, the tree will be destroyed from the left and the right; in which will delete the whole tree. Recursively calling the methods will make the Big O come out to be  $O(\log N)$ .

**Analysis:**  $O(\log N)$

```

void btree::insert(int key) {
    if (root != NULL) {
        insert(key, root);
    }
    else {
        root = new node;
        root->value = key;
        root->left = NULL;
        root->right = NULL;
    }
    // Analysis: O(Log N)
}

```

Here is the **add/insert function** for the Binary Search Tree.

**Precondition:** Check if the root is not a NULL

**Postcondition:** By taking in the key value as the argument, the value will be added to either right or left of the root. The recursive call makes the Big O to come out to be  $O(\log N)$ .

**Analysis:**  $O(\log N)$

```

node* btree::search(int key, node* leaf) {
    if (leaf != NULL) {
        if (key == leaf->value) {
            return leaf;
        }
        if (key < leaf->value) {
            return search(key, leaf->left);
        }
        else {

```



```

        return search(key, leaf->right);
    }
}
else {
    return NULL;
}
}

node* btree::search(int key) {
    return search(key, root);
// Analysis: O(Log N)
}

```

Here is the **search function** for the Binary Search Tree.

**Precondition:** Check if the root is not a NULL

**Postcondition:** In the main, I called the search(int key) method because the leaf was already a tree created. So, I just inputted what value to search for, in which is the key. In that search method, there is a recursive call using both the key and root. By doing so, the search(int key, node\* leaf) method is called. Inside that method there is a check to see if the leaf is NULL, else the search is activated. Once the search is activated, recursion is used to find the root by going either left or right. Using recursion will make the Big O to come out to O(Log N).

**Analysis:** O(Log N)

## Heap:

The heap has the same Big O as the binary search tree for all functions: **add/insert (push), delete (pop), and search**. So the Big O for a Heap is O(log N) for all the functions because of recursive calls and the priority queue. The heap is different from a tree in structure because it is a priority queue and it can be classified as either a max heap or min heap. Min heap is where parent key > child key and a max heap is where parent < child key. To see the code, look in the appendix to see more on how the Heap works.

```

void push(int key)
{
    // insert the new element to the end of the vector
    A.push_back(key);

    // get element index and call heapify-up procedure
    int index = size() - 1;
    heapify_up(index);
// Analysis: O(Log N)
}

// function to remove element with highest priority (present at root)

void pop()
{

```

```

try {
    // if heap has no elements, throw an exception
    if (size() == 0)
        throw out_of_range("Vector<X>::at() : "
                           "index is out of range(Heap underflow)");

    // replace the root of the heap with the last element
    // of the vector
    A[0] = A.back();
    A.pop_back();

    // call heapify-down on root node
    heapify_down(0);
}
// catch and print the exception
catch (const out_of_range & oor) {
    cout << "\n" << oor.what();
}
// Analysis: O(Log N)
}

```

**Table of Data Structures (Average Case):**

Data Structure	Add	Insertion	Delete	Search	Space Complexity
Array	O(1)	O(N)	O(N)	O(N)	O(N)
Linked List	O(1)	O(1)	O(1)	O(N)	O(N)
Stack	O(1)	O(1)	O(1)	O(N)	O(N)
Queue	O(1)	O(1)	O(1)	O(N)	O(N)
Binary Search Trees	O(1)	O(log(N))	O(log(N))	O(log(N))	O(N)
Heap	O(1)	O(N log(N))	O(N log(N))		O(N)
Binary Search	NA	NA	NA	O(Log N)	O(1)
Linear Search	NA	NA	NA	O(N)	O(1)
Insertion Sort	NA	(Sort:) O(N <sup>2</sup> )	NA	NA	O(1)
Merge Sort	NA	(Sort:) O(N)	NA	NA	O(N)

**Table of Data Structures (Worst Case):**

Data Structure	Add	Insertion	Delete	Search	Space Complexity
Array	$O(1)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Linked List	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$
Stack	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$
Queue	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$
Binary Search Trees	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N \log(N))$
Heap	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$		$O(N \log(N))$
Binary Search	NA	NA	NA	$O(\log N)$	$O(1)$
Linear Search	NA	NA	NA	$O(N)$	$O(1)$
Insertion Sort	NA	(Sort:) $O(N^2)$	NA	NA	$O(1)$
Merge Sort	NA	(Sort:) $O(N \log(N))$	NA	NA	$O(N)$

Works Cited Page:

“A Beginner's Guide to Big O Notation.” *A Beginner's Guide to Big O Notation* - Rob Bell, rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/.

Admin. “Min Heap and Max Heap Implementation in C .” *Techie Delight*, 3 June 2019, www.techiedelight.com/min-heap-max-heap-implementation-c/.

Admin. “Stack Implementation in C .” *Techie Delight*, 30 Nov. 2019, www.techiedelight.com/stack-implementation-in-cpp/.

Admin. “Queue Implementation in C .” *Techie Delight*, 30 Nov. 2019, www.techiedelight.com/queue-implementation-cpp/.

“Analysis of Algorithms: Set 2 (Worst, Average and Best Cases).” *GeeksforGeeks*, 21 Dec. 2018, www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/.

“Big O for Beginners.” By, hackernoon.com/big-o-for-beginners-622a64760e2.

“Binary Tree Implementation.” *Binary Tree Implementation*, GitHub, gist.github.com/toboqus/def6a6915e4abd66e922.

*C Program to Implement Insertion Sort*, www.tutorialspoint.com/cplusplus-program-to-implement-insertion-sort.

Daniel Miessler, and Daniel Miessler. “Big-O Notation Explained.” *Daniel Miessler*, danielmiessler.com/study/big-o-notation/.

Manish. "C Program to Implement Merge Sort." *Sanfoundry*, 6 July 2017,

[www.sanfoundry.com/cpp-program-implement-merge-sort/](http://www.sanfoundry.com/cpp-program-implement-merge-sort/).

"Time Complexity vs. Space Complexity." *Educative*, [www.educative.io/edpresso/time-complexity-vs-space-complexity](http://www.educative.io/edpresso/time-complexity-vs-space-complexity).

## Appendix:

### Arrays + Insertion Sort + Linear Search + Binary Search + Merge Sort:

Source Code:

array.h:

```
#pragma once

class Array
{
    int index;
    int capacity;
    int* array;
    Array();

    public:
        Array(int size);
        ~Array();
        bool search( int data);
        void add(int data);
        void insert(int idx, int data);
        void delete_Element (int idx);
        void insertionSort();
        void binarySearch(int target);
        void printArray();
        void Merge( int low, int high, int mid);
        void MergeSort( int p, int r);
};
```

array.cpp:

```
#include <iostream>
#include "array.h"
#include <cassert>
using namespace std;

Array::Array()
{
    index = 0;
    capacity = 0;
}
Array::Array(int size)
{
    capacity = size;
    index = 0;
    array = new int[capacity];
}
Array::~~Array()
{
    delete[] array;
}
```

```

bool Array::search( int data)
{
    for (int i = 0; i < index; i++)
    {
        if (data == array[i])
        {
            return true; // Item Found
        }
    }
    return false; // Item Not Found

    //Analysis: O(N)
}

void Array::add(int data)
{
    if (index + 1 > capacity)
    {
        capacity++;
        int* tmpArray = new int[capacity];
        for (int i = 0; i < index; i++)
        {
            tmpArray[i] = array[i];
        }

        delete array;
        array = tmpArray;
    }

    array[index] = data;
    index++;
    //Analysis: O(1) if the index index + 1 < capacity; else, effeciency is O(N)
}

void Array::insert(int idx, int data)
{
    if (index + 1 > capacity)
    {
        capacity++;
        int* tmpArray = new int[capacity];
        for (int i = 0; i < index; i++)
        {
            tmpArray[i] = array[i];
        }

        delete array;
        array = tmpArray;
    }

    for (int x = index; x >= idx; x--) {
        array[x+1] = array[x];
    }
    array[idx] = data;
    index++;
    //Analysis: O(N)
}

void Array::delete_Element(int idx)
{
    assert(index != 0);
    for (int x = idx; x < index; x++)

```

```

        {
            array[x] = array[x + 1];
        }
        index--;

        //Analysis: O(N)
    }
    void Array::insertionSort() // https://www.tutorialspoint.com/cplusplus-program-to-implement-insertion-sort
    {
        int key, j;
        for (int i = 1; i < index; i++) {
            key = array[i]; //take value
            j = i;
            while (j > 0 && array[j - 1] > key) {
                array[j] = array[j - 1];
                j--;
            }
            array[j] = key; //insert in right place
        }
    }
    void Array::binarySearch(int target) //https://beginnersbook.com/2017/12/c-program-for-binary-search/
    {
        int first = 0;
        int last = index - 1;
        int middle = (first + last) / 2;
        while (first <= last)
        {
            if (array[middle] < target)
            {
                first = middle + 1;
            }
            else if (array[middle] == target)
            {
                cout << target << " found in the array at the location " << middle +
1 << "\n";
                break;
            }
            else {
                last = middle - 1;
            }
            middle = (first + last) / 2;
        }
        if (first > last)
        {
            cout << target << " not found in the array";
        }
    }

    // A function to merge the two half into a sorted data.
    void Array::Merge(int low, int high, int mid)
    {
        // We have low to mid and mid+1 to high already sorted.
        int i, j, k;
        int* temp = new int[high - low + 1];
        i = low;

```



```

k = 0;
j = mid + 1;

// Merge the two parts into temp[].
while (i <= mid && j <= high)
{
    if (array[i] < array[j])
    {
        temp[k] = array[i];
        k++;
        i++;
    }
    else
    {
        temp[k] = array[j];
        k++;
        j++;
    }
}

// Insert all the remaining values from i to mid into temp[].
while (i <= mid)
{
    temp[k] = array[i];
    k++;
    i++;
}

// Insert all the remaining values from j to high into temp[].
while (j <= high)
{
    temp[k] = array[j];
    k++;
    j++;
}

// Assign sorted data stored in temp[] to a[].
for (i = low; i <= high; i++)
{
    array[i] = temp[i - low];
}
delete[] temp;
}

// A function to split array into two parts.
void Array::MergeSort(int low, int high)
{
    int mid;
    if (low < high)
    {
        mid = (low + high) / 2;
        // Split the data into two half.
        MergeSort(low, mid);
        MergeSort(mid + 1, high);

        // Merge them to get sorted output.
        Merge(low, high, mid);
    }
}

```

```

    }
}

void Array::printArray()
{
    for (int i = 0; i < index; i++)
    {
        cout << array[i] << " ";
    }
    //Analysis: O(N)
}

```

### main.cpp:

```

#include "array.h"
#include <iostream>
using namespace std;
int main()
{
    int size = 6;
    //Test 1 :: Add Funtcion
    Array* a1 = new Array(size);
    cout << "Adding to an array: ";
    a1->add(3);
    a1->add(2);
    a1->add(48);
    a1->add(48);
    a1->add(0); //Last Element
    // Resize occurs here:
    a1->add(0);
    a1->add(0);
    a1->printArray();

    cout << endl;

    //Test 2 :: insert Function
    Array* a2 = new Array(size);
    cout << "Inserting to an array: ";
    a2->add(35);
    a2->add(75);
    a2->add(48);
    a2->insert(1, 69);
    a2->printArray();
    cout << endl;

    //Test 3 :: Search Function
    if (a2->search(10) == true)
    {
        cout << "Item was Found!" << endl;
    }
    else
    {
        cout << "Item wasn't Found!" << endl;
    }
}

```

```

//Test 4:: Delete Function
a2->delete_Element(2);
cout << "Deleting 75: ";
a2->printArray();
cout << endl;

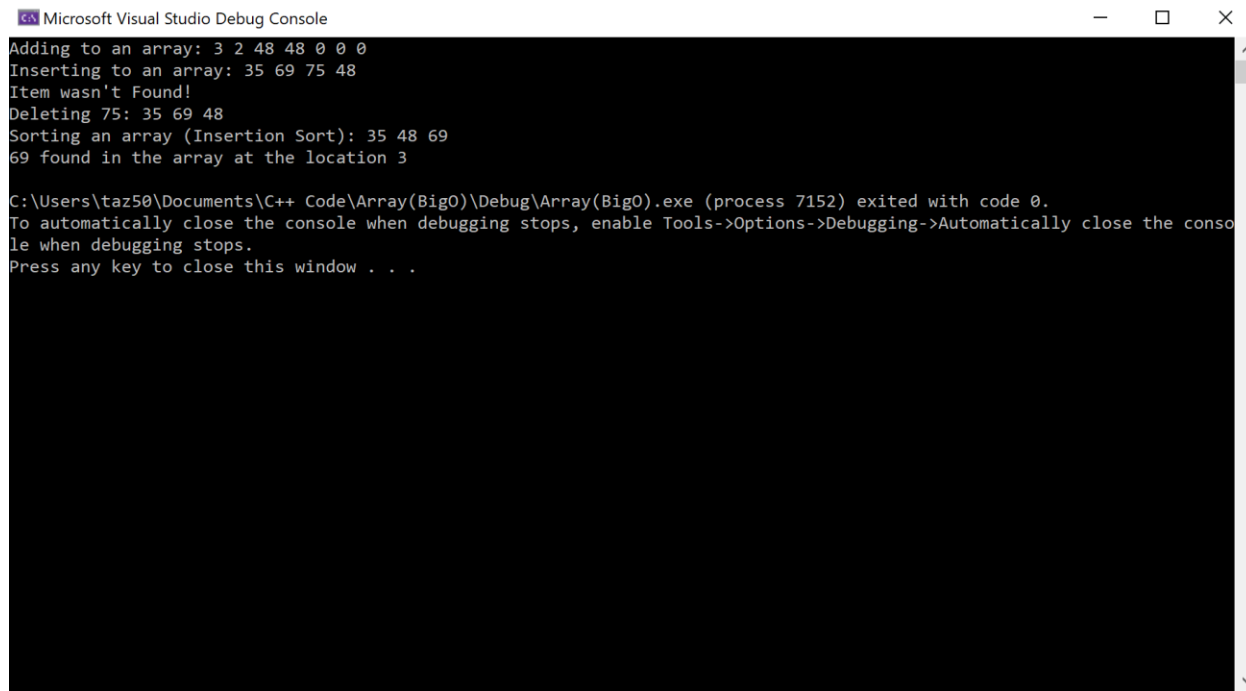
//Test 5:: Insertion Sort
a2->insertionSort();
cout << "Sorting an array (Insertion Sort): ";
a2->printArray();

//Test 6:: Binary Search
cout << endl;
a2->binarySearch(69);

cout << "Sorting an array (Merge Sort): ";
a1->MergeSort(0, size);
a1->printArray();
delete a1;
delete a2;
}

```

### Screenshot (Without Merge Sort):



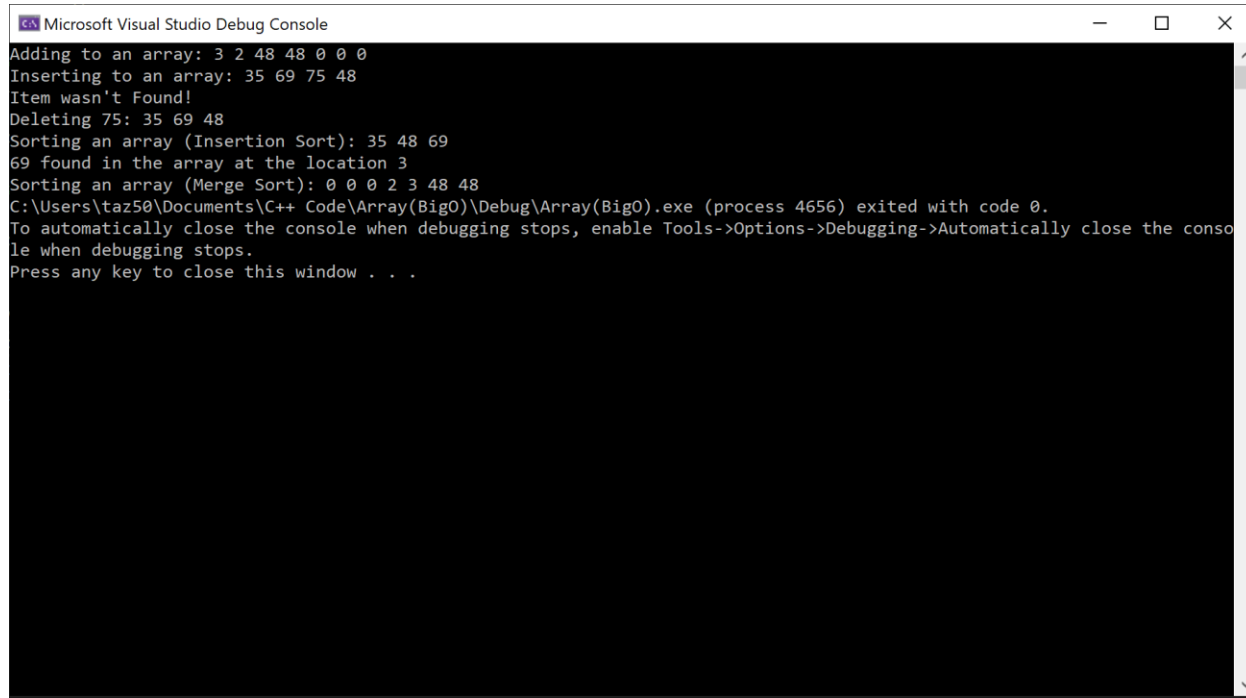
The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output is as follows:

```

Adding to an array: 3 2 48 48 0 0 0
Inserting to an array: 35 69 75 48
Item wasn't Found!
Deleting 75: 35 69 48
Sorting an array (Insertion Sort): 35 48 69
69 found in the array at the location 3

C:\Users\taz50\Documents\C++ Code\Array(Big0)\Debug\Array(Big0).exe (process 7152) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Screenshot (With Merge Sort):


```

Microsoft Visual Studio Debug Console
Adding to an array: 3 2 48 48 0 0 0
Inserting to an array: 35 69 75 48
Item wasn't Found!
Deleting 75: 35 69 48
Sorting an array (Insertion Sort): 35 48 69
69 found in the array at the location 3
Sorting an array (Merge Sort): 0 0 0 2 3 48 48
C:\Users\taz50\Documents\C++ Code\Array(Big0)\Debug\Array(Big0).exe (process 4656) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Linked List:toolkit.h:

```

#pragma once
#ifndef TOOLKIT_H
#define TOOLKIT_H

#include <cstdlib>

class node {
public:
    // TYPEDEF
    typedef double value_type;

    // CONSTRUCTOR
    node(
        const value_type& init_data = value_type(),
        node* init_link = NULL
    ) {
        data_field = init_data;
        link_field = init_link;
    }

```

```

    }

    // Member functions to set the data and link fields:
    void set_data(const value_type& new_data) { data_field = new_data; }

    void set_link(node* new_link) { link_field = new_link; }

    // Constant member function to retrieve the current data:
    value_type data() const { return data_field; }

    // Two slightly different member functions to retrieve
    // the current link:
    const node* link() const { return link_field; }

    node* link() { return link_field; }

private:
    value_type data_field;
    node* link_field;
};

// FUNCTIONS for the linked list toolkit
std::size_t list_length(const node* head_ptr);

void list_head_insert(node*& head_ptr, const node::value_type& entry);

void list_insert(node* previous_ptr, const node::value_type& entry);

node* list_search(node* head_ptr, const node::value_type& target);

const node* list_search
(const node* head_ptr, const node::value_type& target);

node* list_locate(node* head_ptr, std::size_t position);

const node* list_locate(const node* head_ptr, std::size_t position);

void list_head_remove(node*& head_ptr);

void list_remove(node* previous_ptr);

void list_clear(node*& head_ptr);

void list_copy(const node* source_ptr, node*& head_ptr, node*& tail_ptr);

#endif // TOOLKIT_H

```

### toolkit.cpp

```

#include "toolkit.h"
#include <cstdlib>
#include <assert.h>

std::size_t list_length(const node* head_ptr) {
    int length = 0;

```

```

    const node* trav = head_ptr;
    while (trav != NULL) {
        length++;
        trav = trav->link();
    }
    return length;
}

void list_head_insert(node*& head_ptr, const node::value_type& entry) {
    node* new_node = new node();
    new_node->set_data(entry);
    new_node->set_link(head_ptr);
    head_ptr = new_node;
}

void list_insert(node* previous_ptr, const node::value_type& entry) {
    node* new_node = new node(entry);
    new_node->set_link(previous_ptr->link());
    previous_ptr->set_link(new_node);
}

node* list_search(node* head_ptr, const node::value_type& target) {
    node* trav = head_ptr;
    // the first part of the condition guards the second part
    while ((trav != NULL) && (trav->data() != target)) {
        trav = trav->link();
    }
    // examine the results
    // if trav is NULL, we didn't find it, so return NULL
    // if trav is not NULL then trav->data() == target, so return trav
    return trav;
}

void list_head_remove(node*& head_ptr) {
    assert(head_ptr != NULL);
    node* tmp = head_ptr; // first node, soon to be removed
    head_ptr = head_ptr->link(); // head_ptr now points at the second node
    delete tmp;
}

void list_remove(node* previous_ptr)
{
    node* remove_ptr;

    remove_ptr = previous_ptr->link();
    previous_ptr->set_link(remove_ptr->link());
    delete remove_ptr;
}

node* list_locate(node* head_ptr, std::size_t position) {
    if (position == 1) {
        return head_ptr;
    }
    else if (head_ptr == NULL) {

```

```

        return NULL;
    }
    else {
        return list_locate(head_ptr->link(), position - 1);
    }
}

const node* list_locate(const node* head_ptr, std::size_t position) {
    const node* trav = head_ptr;
    int count = position;
    while ((trav != NULL) && (count > 1)) {
        trav = trav->link();
        count--;
    }
    return trav;
}

void list_copy(const node* source_ptr, node*& head_ptr, node*& tail_ptr)
// Library facilities used: cstdlib
{
    head_ptr = NULL;
    tail_ptr = NULL;

    // Handle the case of the empty list.
    if (source_ptr == NULL)
        return;

    // Make the head node for the newly created list, and put data in it.
    list_head_insert(head_ptr, source_ptr->data());
    tail_ptr = head_ptr;

    // Copy the rest of the nodes one at a time, adding at the tail of new list.
    source_ptr = source_ptr->link();
    while (source_ptr != NULL)
    {
        list_insert(tail_ptr, source_ptr->data());
        tail_ptr = tail_ptr->link();
        source_ptr = source_ptr->link();
    }
}

void list_clear(node*& head_ptr) {
    while (head_ptr != NULL) {
        // we know that head_ptr is not NULL
        // so head_ptr points at the first node of the rest of the list
        node* rest = head_ptr->link();
        delete head_ptr; // get rid of first node
        head_ptr = rest;
    }
}

```

main.cpp:

```

#include <cassert>
#include <iostream>
#include "toolkit.h"

using namespace std;

void list_print(node* head_ptr)
{
    node* cursor;
    cursor = head_ptr;
    while (cursor != NULL)
    {
        cout << " " << cursor->data();
        cursor = cursor->link();
    }
    cout << endl;
}

int main()
{
    //Test 1://
    node* head_ptr = NULL;
    list_head_insert(head_ptr, 23.5);
    list_head_insert(head_ptr, 45.6);
    list_head_insert(head_ptr, 67.7);
    list_head_insert(head_ptr, 89.8);
    list_head_insert(head_ptr, 12.9);

    cout << "Insert at the head in linked list 1: ";
    list_print(head_ptr);
    cout << endl;

    //Test 2: Insert at a given link() value//
    node* head = NULL;
    node* end = NULL;

    list_head_insert(head, 23.5);

    end = head;
    list_insert(end, 45.6);
    list_insert(end->link(), 67.7);
    end = end->link();
    list_insert(end->link(), -123.5);
    end = end->link();
    list_insert(end->link(), 89.8);
    end = end->link();
    list_insert(end->link(), 12.9);

    cout << "Inserting at the Tail of the list in linked list 2: ";
    list_print(head);

```



```
cout << endl;

//Test 3://
cout << "Remove at the Head in linked list 2: ";
list_head_remove(head);
list_print(head);
cout << endl;

//Test 4://
node* cursor = head;
cursor = cursor->link();
list_remove(cursor);
cout << "Remove -123.5 in linked list 2: ";
list_print(head);
cout << endl;

//Test 5: Search Function//
if (list_search(head_ptr, 23.5) != NULL)
{
    cout << "Item was found in linked list 1!";
}
else
{
    cout << "Item wasn't found in linked list 1!";
}
}
```

Screenshot:

```

Microsoft Visual Studio Debug Console
Insert at the head in linked list 1: 12.9 89.8 67.7 45.6 23.5

Inserting at the Tail of the list in linked list 2: 23.5 45.6 67.7 -123.5 89.8 12.9

Remove at the Head in linked list 2: 45.6 67.7 -123.5 89.8 12.9

Remove -123.5 in linked list 2: 45.6 67.7 89.8 12.9

Item was found in linked list 1!
C:\Users\taz50\Documents\C++ Code\LinkedList(Big0)\Debug\LinkedList(Big0).exe (process 6184) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

## Stack:

### main.cpp

```

#include <iostream>
#include <cstdlib>
using namespace std;

// define default capacity of the stack
#define SIZE 10

// Class for stack
class stack //https://www.techiedelight.com/stack-implementation-in-cpp/
{
    int* arr;
    int top;
    int capacity;

public:
    stack(int size = SIZE);           // constructor
    ~stack();                         // destructor

    void push(int);
    int pop();
    int peek();

    int size();
    bool isEmpty();

```

```

        bool isFull();
        bool search(int target);
};

// Constructor to initialize stack
stack::stack(int size)
{
    arr = new int[size];
    capacity = size;
    top = -1;
}

// Destructor to free memory allocated to the stack
stack::~~stack()
{
    delete arr;
}

// Utility function to add an element x in the stack
void stack::push(int x)
{
    if (isFull())
    {
        cout << "OverFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << x << endl;
    arr[++top] = x;
}

// Utility function to pop top element from the stack
int stack::pop()
{
    // check for stack underflow
    if (isEmpty())
    {
        cout << "UnderFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Removing " << peek() << endl;

    // decrease stack size by 1 and (optionally) return the popped element
    return arr[top--];
}

// Utility function to return top element in a stack
int stack::peek()
{
    if (!isEmpty())
        return arr[top];
    else
        exit(EXIT_FAILURE);
}

// Utility function to return the size of the stack
int stack::size()

```

```

{
    return top + 1;
}

// Utility function to check if the stack is empty or not
bool stack::isEmpty()
{
    return top == -1;    // or return size() == 0;
}

// Utility function to check if the stack is full or not
bool stack::isFull()
{
    return top == capacity - 1; // or return size() == capacity;
}

bool stack::search(int target)
{
    for (size_t i = 0; i < capacity; i++)
    {
        if (target == arr[i])
        {
            return true;
        }
    }
    return false;
}

// main function
int main()
{
    stack pt(3);

    pt.push(1);
    pt.push(2);

    pt.pop();
    pt.pop();

    pt.push(3);

    cout << "Top element is: " << pt.peek() << endl;

    cout << "Stack size is " << pt.size() << endl;

    pt.pop();

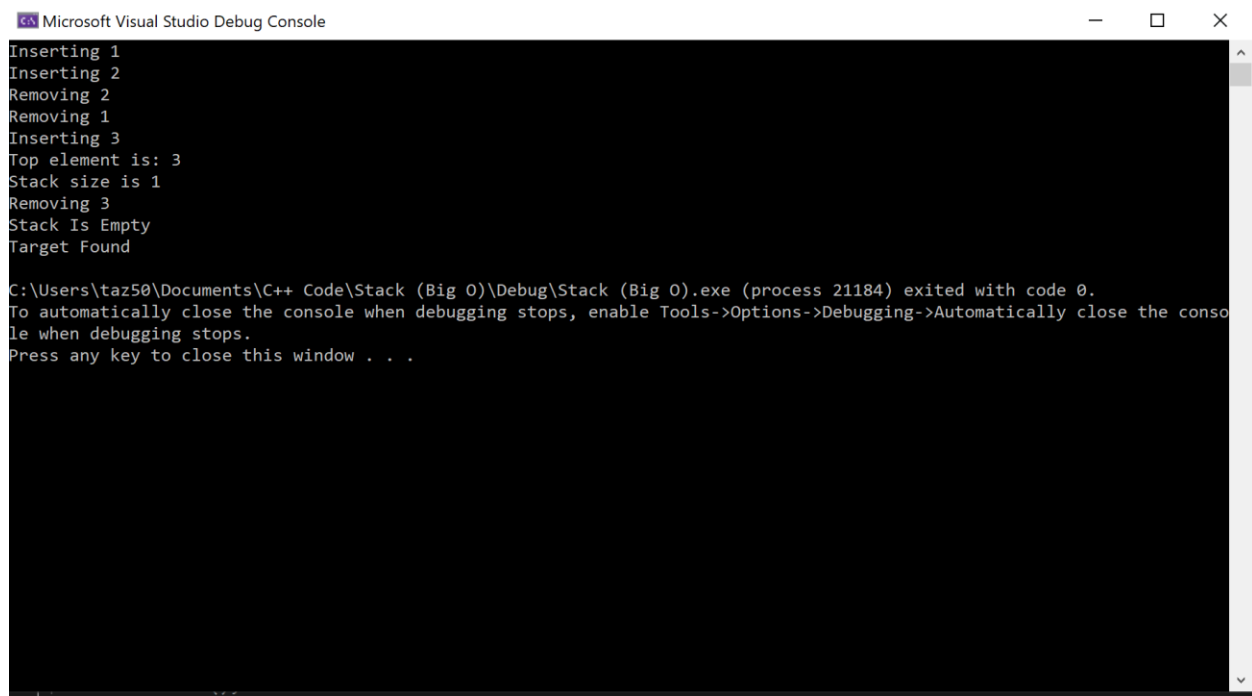
    if (pt.isEmpty())
        cout << "Stack Is Empty\n";
    else
        cout << "Stack Is Not Empty\n";

    if(pt.search(3) == true)
        cout << "Target Found\n";
    else
        cout << "Target Not Found\n";
}

```

```
    return 0;
}
```

### Screenshot:



```
Microsoft Visual Studio Debug Console
Inserting 1
Inserting 2
Removing 2
Removing 1
Inserting 3
Top element is: 3
Stack size is 1
Removing 3
Stack Is Empty
Target Found

C:\Users\taz50\Documents\C++ Code\Stack (Big O)\Debug\Stack (Big O).exe (process 21184) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

## Queue:

### main.cpp:

```
#include <iostream>
#include <cstdlib>
using namespace std;

// define default capacity of the queue https://www.techiedelight.com/queue-implementation-cpp/
#define SIZE 10

// Class for queue
class queue
{
    int* arr;           // array to store queue elements
    int capacity;       // maximum capacity of the queue
    int front;          // front points to front element in the queue (if any)
    int rear;           // rear points to last element in the queue
    int count;          // current size of the queue

public:
    queue(int size = SIZE);           // constructor
    ~queue();                         // destructor

    void dequeue();
    void enqueue(int x);
    int peek();
```

```

        int size();
        bool isEmpty();
        bool isFull();
        bool search(int target);
};

// Constructor to initialize queue
queue::queue(int size)
{
    arr = new int[size];
    capacity = size;
    front = 0;
    rear = -1;
    count = 0;
}

// Destructor to free memory allocated to the queue
queue::~queue()
{
    delete arr;
}

// Utility function to remove front element from the queue
void queue::dequeue()
{
    // check for queue underflow
    if (isEmpty())
    {
        cout << "UnderFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Removing " << arr[front] << '\n';

    front = (front + 1) % capacity;
    count--;
}

// Utility function to add an item to the queue
void queue::enqueue(int item)
{
    // check for queue overflow
    if (isFull())
    {
        cout << "OverFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << item << '\n';

    rear = (rear + 1) % capacity;
    arr[rear] = item;
    count++;
}

// Utility function to return front element in the queue
int queue::peek()
{

```

```

        if (isEmpty())
        {
            cout << "UnderFlow\nProgram Terminated\n";
            exit(EXIT_FAILURE);
        }
        return arr[front];
    }

bool queue::search(int target)
{
    for (size_t i = 0; i < capacity; i++)
    {
        if (arr[i] == target)
            return true;
    }
    return false;
}

// Utility function to return the size of the queue
int queue::size()
{
    return count;
}

// Utility function to check if the queue is empty or not
bool queue::isEmpty()
{
    return (size() == 0);
}

// Utility function to check if the queue is full or not
bool queue::isFull()
{
    return (size() == capacity);
}

// main function
int main()
{
    // create a queue of capacity 5
    queue q(5);

    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << "Front element is: " << q.peek() << endl;
    q.dequeue();

    q.enqueue(4);

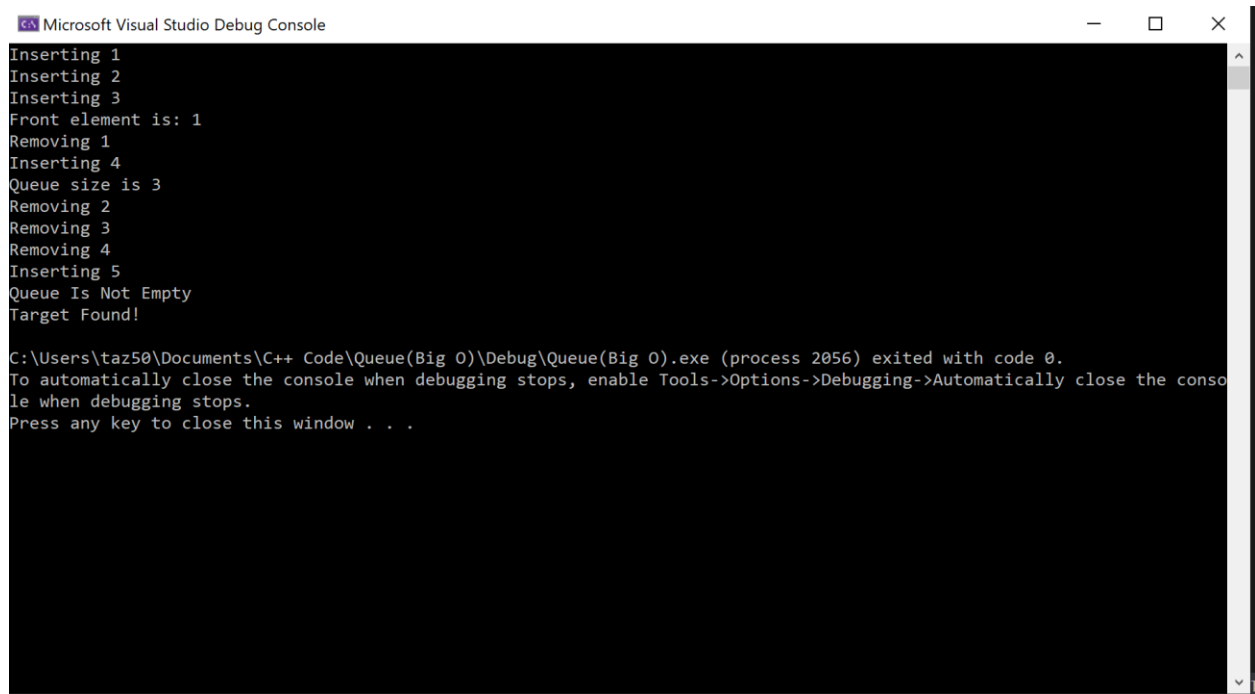
    cout << "Queue size is " << q.size() << endl;

    q.dequeue();
    q.dequeue();
    q.dequeue();
    q.enqueue(5);
}

```

```
    if (q.isEmpty())  
        cout << "Queue Is Empty\n";  
    else  
        cout << "Queue Is Not Empty\n";  
  
    if (q.search(5) == true)  
        cout << "Target Found!\n";  
    else  
        cout << "Target not found!\n";  
  
    return 0;  
}
```

### Screenshot:

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the Visual Studio logo and the text "Microsoft Visual Studio Debug Console". The console output is as follows:  
Inserting 1  
Inserting 2  
Inserting 3  
Front element is: 1  
Removing 1  
Inserting 4  
Queue size is 3  
Removing 2  
Removing 3  
Removing 4  
Inserting 5  
Queue Is Not Empty  
Target Found!  
  
C:\Users\taz50\Documents\C++ Code\Queue(Big 0)\Debug\Queue(Big 0).exe (process 2056) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .  
The console window has a scrollbar on the right side, and the text is white on a black background.



## Binary Search Tree:

main.cpp:

```
#include <iostream>

using namespace std;

struct node {
    int value;
    node* left;
    node* right;
};

class btree {
public:
    btree();
    ~btree();

    void insert(int key);
    node* search(int key);
    void destroy_tree();
    void inorder_print();
    void postorder_print();
    void preorder_print();

private:
    void destroy_tree(node* leaf);
    void insert(int key, node* leaf);
    node* search(int key, node* leaf);
    void inorder_print(node* leaf);
    void postorder_print(node* leaf);
    void preorder_print(node* leaf);

    node* root;
};

btree::btree() {
    root = NULL;
}

btree::~btree() {
    destroy_tree();
}

void btree::destroy_tree(node* leaf) {
    if (leaf != NULL) {
        destroy_tree(leaf->left);
        destroy_tree(leaf->right);
        delete leaf;
    }
}
```

```

}

void btree::insert(int key, node* leaf) {
    if (key < leaf->value) {
        if (leaf->left != NULL) {
            insert(key, leaf->left);
        }
        else {
            leaf->left = new node;
            leaf->left->value = key;
            leaf->left->left = NULL;
            leaf->left->right = NULL;
        }
    }
    else if (key >= leaf->value) {
        if (leaf->right != NULL) {
            insert(key, leaf->right);
        }
        else {
            leaf->right = new node;
            leaf->right->value = key;
            leaf->right->right = NULL;
            leaf->right->left = NULL;
        }
    }
}

}

void btree::insert(int key) {
    if (root != NULL) {
        insert(key, root);
    }
    else {
        root = new node;
        root->value = key;
        root->left = NULL;
        root->right = NULL;
    }
}

}

node* btree::search(int key, node* leaf) {
    if (leaf != NULL) {
        if (key == leaf->value) {
            return leaf;
        }
        if (key < leaf->value) {
            return search(key, leaf->left);
        }
        else {
            return search(key, leaf->right);
        }
    }
    else {
        return NULL;
    }
}

}

```

```

node* btree::search(int key) {
    return search(key, root);
}

void btree::destroy_tree() {
    destroy_tree(root);
}

void btree::inorder_print() {
    inorder_print(root);
    cout << "\n";
}

void btree::inorder_print(node* leaf) {
    if (leaf != NULL) {
        inorder_print(leaf->left);
        cout << leaf->value << ",";
        inorder_print(leaf->right);
    }
}

void btree::postorder_print() {
    postorder_print(root);
    cout << "\n";
}

void btree::postorder_print(node* leaf) {
    if (leaf != NULL) {
        inorder_print(leaf->left);
        inorder_print(leaf->right);
        cout << leaf->value << ",";
    }
}

void btree::preorder_print() {
    preorder_print(root);
    cout << "\n";
}

void btree::preorder_print(node* leaf) {
    if (leaf != NULL) {
        cout << leaf->value << ",";
        inorder_print(leaf->left);
        inorder_print(leaf->right);
    }
}

int main() {

    //btree tree;
    btree* tree = new btree();

    tree->insert(10);
    tree->insert(6);
    tree->insert(14);
    tree->insert(5);
    tree->insert(8);
    tree->insert(11);

```

```
tree->insert(18);

node* found = tree->search(6);
if (found != NULL)
    cout << "Item Found" << endl;
else
    cout << "Item Not Found" << endl;
cout << endl;

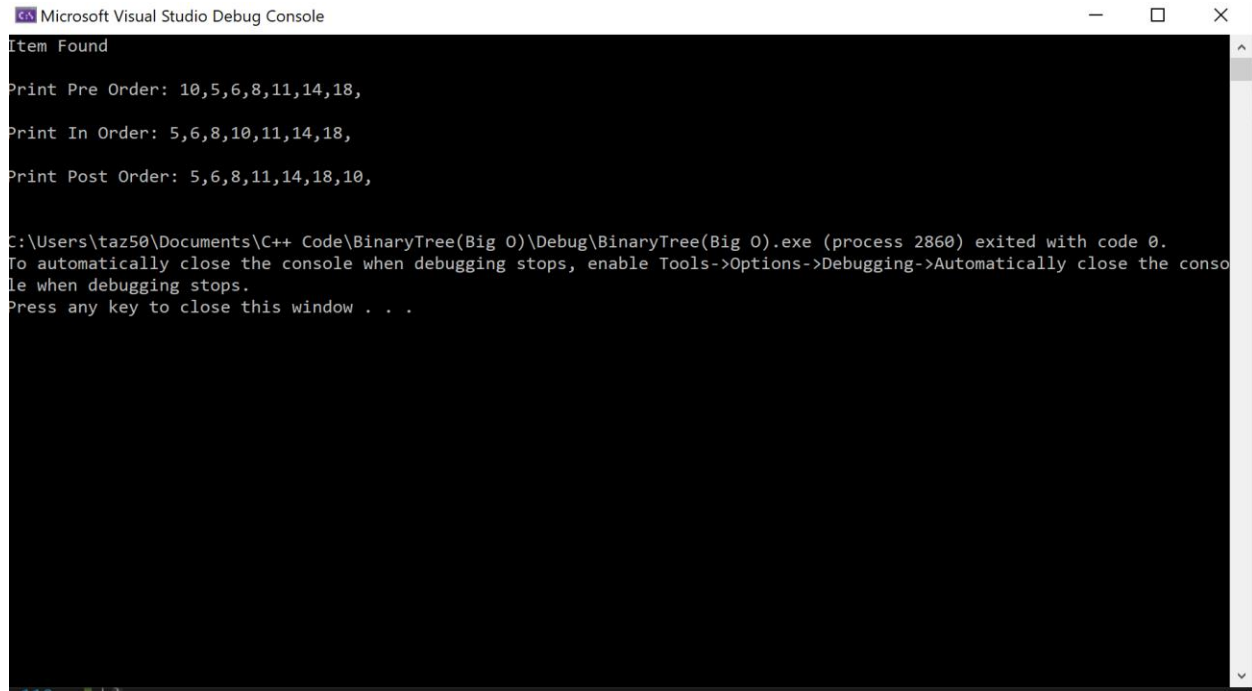
cout << "Print Pre Order: ";
tree->preorder_print();
cout << endl;

cout << "Print In Order: ";
tree->inorder_print();
cout << endl;

cout << "Print Post Order: ";
tree->postorder_print();
cout << endl;

delete tree;
}
```

Screenshot:



```

Microsoft Visual Studio Debug Console

Item Found

Print Pre Order: 10,5,6,8,11,14,18,
Print In Order: 5,6,8,10,11,14,18,
Print Post Order: 5,6,8,11,14,18,10,

C:\Users\taz50\Documents\C++ Code\BinaryTree(Big 0)\Debug\BinaryTree(Big 0).exe (process 2860) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

## Heap:

main.cpp:

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
using namespace std;

// Data structure for Max Heap
struct PriorityQueue
{
private:
    // vector to store heap elements
    vector<int> A;

    // return parent of A[i]
    // don't call this function if it is already a root node
    int PARENT(int i)
    {
        return (i - 1) / 2;
    }

    // return left child of A[i]
    int LEFT(int i)
    {
        return (2 * i + 1);
    }

    // return right child of A[i]

```

```

int RIGHT(int i)
{
    return (2 * i + 2);
}

// Recursive Heapify-down algorithm
// the node at index i and its two direct children
// violates the heap property
void heapify_down(int i)
{
    // get left and right child of node at index i
    int left = LEFT(i);
    int right = RIGHT(i);

    int largest = i;

    // compare A[i] with its left and right child
    // and find largest value
    if (left < size() && A[left] > A[i])
        largest = left;

    if (right < size() && A[right] > A[largest])
        largest = right;

    // swap with child having greater value and
    // call heapify-down on the child
    if (largest != i) {
        swap(A[i], A[largest]);
        heapify_down(largest);
    }
}

// Recursive Heapify-up algorithm
void heapify_up(int i)
{
    // check if node at index i and its parent violates
    // the heap property
    if (i && A[PARENT(i)] < A[i])
    {
        // swap the two if heap property is violated
        swap(A[i], A[PARENT(i)]);

        // call Heapify-up on the parent
        heapify_up(PARENT(i));
    }
}

public:
    // return size of the heap
    unsigned int size()
    {
        return A.size();
    }

    // function to check if heap is empty or not
    bool empty()
    {
        return size() == 0;
    }

```

```

    }

    // insert key into the heap
    void push(int key)
    {
        // insert the new element to the end of the vector
        A.push_back(key);

        // get element index and call heapify-up procedure
        int index = size() - 1;
        heapify_up(index);
    }

    // function to remove element with highest priority (present at root)
    void pop()
    {
        try {
            // if heap has no elements, throw an exception
            if (size() == 0)
                throw out_of_range("Vector<X>::at() : "
                                    "index is out of range(Heap underflow)");

            // replace the root of the heap with the last element
            // of the vector
            A[0] = A.back();
            A.pop_back();

            // call heapify-down on root node
            heapify_down(0);
        }
        // catch and print the exception
        catch (const out_of_range & oor) {
            cout << "\n" << oor.what();
        }
    }

    // function to return element with highest priority (present at root)
    int top()
    {
        try {
            // if heap has no elements, throw an exception
            if (size() == 0)
                throw out_of_range("Vector<X>::at() : "
                                    "index is out of range(Heap underflow)");

            // else return the top (first) element
            return A.at(0); // or return A[0];
        }
        // catch and print the exception
        catch (const out_of_range & oor) {
            cout << "\n" << oor.what();
        }
    }
};

int main()
{
    PriorityQueue pq;

```

```

// Note - Priority is decided by element's value

pq.push(3);
pq.push(2);
pq.push(15);

cout << "Size is " << pq.size() << endl;

cout << pq.top() << " ";
pq.pop();

cout << pq.top() << " ";
pq.pop();

pq.push(5);
pq.push(4);
pq.push(45);

cout << endl << "Size is " << pq.size() << endl;

cout << pq.top() << " ";
pq.pop();

cout << pq.top() << " ";
pq.pop();

cout << pq.top() << " ";
pq.pop();

cout << endl << std::boolalpha << pq.empty();

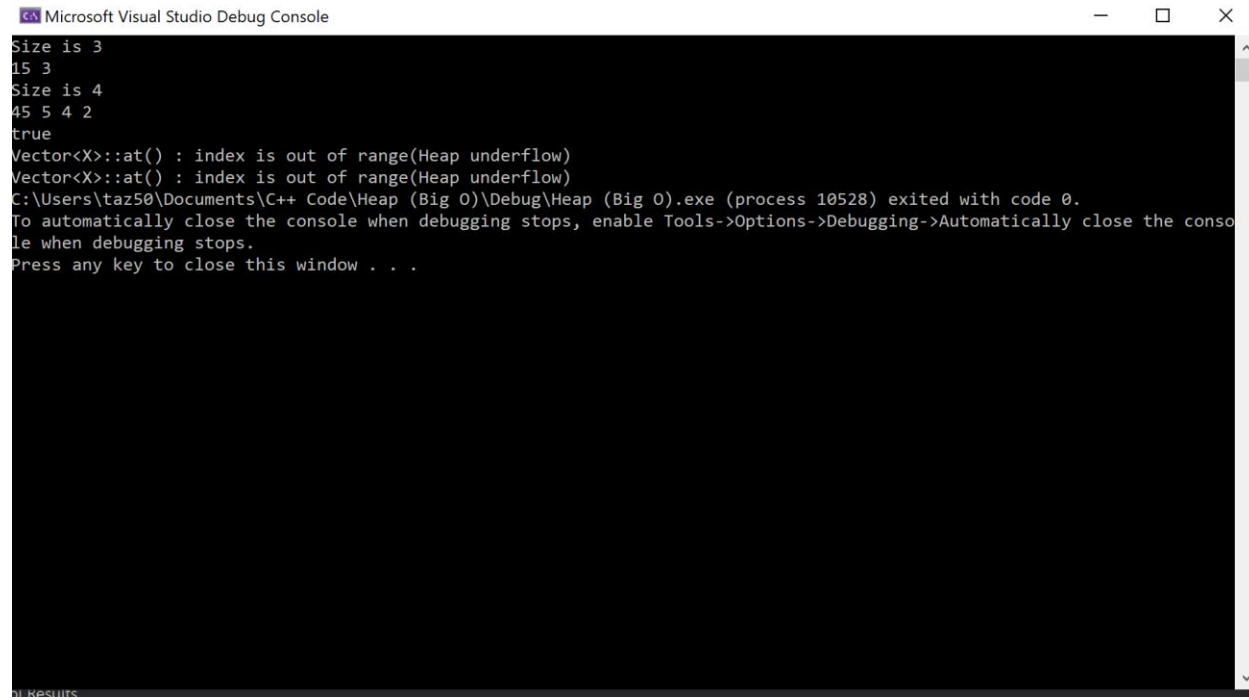
pq.top();    // top operation on an empty heap
pq.pop();    // pop operation on an empty heap

return 0;
}

```

Screenshot:





The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output is as follows:

```
Size is 3
15 3
Size is 4
45 5 4 2
true
Vector<X>::at() : index is out of range(Heap underflow)
Vector<X>::at() : index is out of range(Heap underflow)
C:\Users\taz50\Documents\C++ Code\Heap (Big 0)\Debug\Heap (Big 0).exe (process 10528) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

At the bottom left of the console window, the text "DL Results" is visible.